EEE3027: Final Essay

16th of May 2023
URN: 6596386

School of Computer Science and Electronic Engineering
Faculty of Engineering and Physical Sciences
University of Surrey

Contents

A System on a Chip Interfacing

A)a JTAG Advantages

As chips have become denser, directly probing pins, such as with the old bed-of-nails technique, is no longer possible, yet accessing these pins is vital for testing. This is the problem the Joint Test Action Group set out to solve with JTAG, which is a specification for accessing on-ship resources and boundary-scan hardware testing on the board[1]. Without boundary-scanning, the only test option is functional, which requires massive amounts of test cases to exhaust all possibilities and does not help find the cause of a failure. Meanwhile, JTAG's boundary scan enables one to control and monitor individual IO pins, allowing for structural tests. Structural tests will help find assembly defects, such as shorts, and run much quicker as they are non-exhaustive.

JTAG also offers other benefits, such as monitoring systems during operation without disrupting them and even testing systems that are otherwise non-functional[2]. This is because JTAG is independent of the system and accessible via just four pins (depending on specification). As the logic and pins are part of the IC, JTAG will allow for testing throughout a product's life cycle. Engineers can use the interface during prototyping, production, and after deployment[2]. JTAG also daisy chains, allowing multiple devices within a system to be tested with one interface.

A)b JTAG Functionality

The JTAG implementation shown in Figure 1 of the essay brief connects to the internal bus of the ARM Cortex-M0 SoC. This connection enables the JTAG interface to control and monitor the data on that bus, allowing one to shift in a test pattern to test the NVIC, processor core, debug subsystem, memory and peripherals, or any combination thereof.

This sort of boundary testing is done via a path of interconnected shift-register cells on each signal pin in the bus, which are controlled via the Test Access Port (TAP) of the JTAG. The TAP has four core control signals[3]:

- TCK, clock to synchronize operation
- TMS, mode select line
- TDI, input shifted into cells on the rising edge
- TDO, output shifted out of cells on the falling edge
- (Optional) TRST, a reset for the state machine

Creating this functionality in VHDL is done by creating a state machine, shown in Figure 1, which is controlled by the value of TMS during the rising edge of TCK. The state machine has two paths to capture and control the data on the instruction registers (IR) or the data registers (DR).

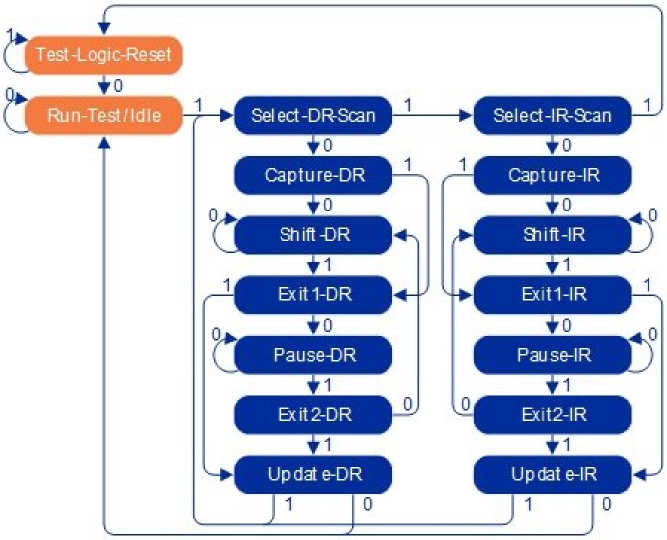

Figure 1: JTAG State Machine Diagram from Corelis[2]

The cells must also be created in VHDL using multiplexers and latches. These cells have a parallel input and output, which allow for the regular operation of the device, and shift inputs and outputs, which are used during tests to load data in or read data out.

A)c VHDL Implementation

Various VHDL implementation concepts can be used to create a low-latency JTAG implementation. As a whole JTAG throughput can be increased via parallelizing and pipeline, which involved doing multiple operations in parallel and breaking one operation into stages respectively. Parallelizing is well suited to combinational logic while pipelining lends itself to sequential logic. However, neither of these method decrease latency instead just increases system throughput. The main areas to optimize for latency in JTAG are the shift registers and the finite state machine driving the JTAG logic.

As mentioned the TAP in JTAG functions with a state machine, which can be optimized in different ways. The finite state machine can be split into next-state logic, state memory, output logic, and output register. Next-state logic has a sensitivity list of only the TMS line, meaning the logic triggers exclusively when the TMS line is updated, and on the rising edge of the clock, its output will be saved as the current state in the state memory. Splitting it like this allows reduces the fan-out and the amount of resources that need to be used. The state can also be stored as a 4-bit integer (as there are 16 states), assuming this system does not need to use one-hot encoding as it is for testing. Finally for the state machine implementing asynchronous reset, meaning that the rest will occur regardless of the clock, will help as it allows quicker resetting between tests.

A lot of the latency in JTAG occurs while data is being shifted into or out of the shift registers (both the instruction and the data registers.) To speed this up a specific type of shift register, the barrel shifter, can be used. Barrel shifters are purely combinational, consisting essentially of sequences of multiplexers, and can shift a specific number of bits over which can be useful for quickly loading and unloading. This would also require some logic on the TDI line as data comes in serial and would need to be made parallel.

JTAG also makes use of BSDL, Boundary Scan Description Language, which is a subset of VHDL used to give accurate descriptions of the features of a device with JTAG (or other boundary scanning methods). The descriptions define each digital signal and can be given to automated tools to generate test programs for the board[2].

A)d Calculations Single Stage

One can calculate throughput, latency, and clock period given time per unit of work and the register overhead. Throughput (Θ) is the number of tasks a module can solve in a given period of time, with a standard unit of mega-operations (millions of operations) per second or mops[4]. Latency (T) is the time taken per task[4]. These two values are reciprocal ($\Theta = \frac{1}{T}$). The clock period (T_{clk}) is the time from one rising edge to the next

For a generic single-stage processor, which takes 24 ns per instruction (T_m), and with flip-flops at the input and output, with 300 ps overhead(t_{reg}), the following values can be calculated:

$$T = T_m + t_{\text{reg}} = 24 \text{ ns} + .3 \text{ ns} = 24.3 \text{ ns} \tag{1}$$

$$\Theta = \frac{1}{T} = \frac{1}{24.3} = 41.15 \text{ Mops} \tag{2}$$

$$T_{\text{clk}} = 24.3 \text{ ns} \tag{3}$$

This system area usage could also be calculated if the area of the module (a_m) and the area of the register (a_{ref}) were known, using the following equation.

$$a = a_m + a_{\text{ref}} \tag{4}$$

(133)

A)e Calculations Four Core

This part expands on part A)d with a four-core system with nine stages. To create such a system the original module must both be pipelined and parallelized. Pipelining involves taking a large operation and breaking it into submodules, this means that submodules will not be idle long as they can begin processing new data as soon as they finish the previous task. This helps increase throughput at the cost of needing registers between the submodules. Parallelizing simply duplicates the entire module to allow multiple operations to be done at the same time. This will increase throughput by a factor of how many parallel systems exist. However, this comes at the cost of needing to replicate the entire module

For the given example, to find the new throughput, latency, and clock period, the effect of using nine stages ($n = 9$) must first be calculated. This has to account for the added register overhead[4] (T_m and t_{reg} are unchanged).

$$T = T_m + nt_{\text{reg}} = 24 \text{ ns} + 9 * .3 \text{ ns} = 26.7 \text{ ns} \tag{5}$$

$$\Theta_{\text{pipeline}} = \frac{1}{T_m/n + t_{\text{reg}}} = \frac{1}{24 \text{ ns}/9 + .3 \text{ ns}} = \frac{1}{2.967 \text{ ns}} = 337.08 \text{ Mops} \tag{6}$$

$$T_{\text{clk}} = 26.7 \text{ ns}/9 = 2.967 \text{ ns} \tag{7}$$

This nine-stage pipelined process can now be parallelized. To do this it is assumed that the module can be split into nine equal parts. The four-core nature of the new system does not add any timing overhead but will increase throughput four-fold, as the task will be completed in parallel.

$$\Theta = \Theta_{\text{pipeline}} * 4 = 337.08 \text{ Mops} * 4 = 1348.31 \text{ Mops} \tag{8}$$

These changes make the new system 32 times faster than the one in part A)d. The cost of this new design's performance is a considerable increase in area. The new requirement can be calculated as follows, with n being the number of stages and p the number of parallel systems:

$$a = p(a_m + na_{\text{ref}}) \tag{9}$$

$$a = 4a_m + 36a_{\text{ref}} \tag{10}$$

Pipelining must also be done in a manner that avoids bottlenecking the system, as the slowest module limits throughput.

B Design Challenge

B)a Architecture and Block's Design

For this section, an AMBA interface to six external SRAM devices with a Hamming encoder and decoder was to be designed. This will allow the processor in Figure 1 of the essay brief to use SRAM with some extra noise protection from the Hamming codes. To do this the following assumptions will be made, firstly it is assumed that the data to be stored on the SRAM is in 16-bit words. It will also be assumed that this system is non-critical and that noise due to environmental factors (radiation, EMI, etc.) is minimal. Finally, it is assumed that SRAM devices consist of just memory cells and require an external controller. With the assumptions the design in Figure 2 was created.

Figure 2: Architecture of AMBA Interface

Before looking at the individual blocks AMBA, or Advanced Microcontroller Bus Architecture, must be discussed. AMBA is an open architecture used as the standard solution for how blocks on an SoC or ASIC interface[5]. The Cortex-M design uses the AHB-Lite (High-performance Bus) protocol, which is a version of AHB with a single master. With this knowledge, a module provided by ARM, the creators of AMBA, for communicating between an AHB and an external SRAM called `ahb_to_extmem16.v` can be used[6]. The pinout of this block can be seen in Figure 3, which shows how the information from the bus gets converted into the required inputs and outputs for memory. The important outputs of the block are address, data, and write line, with the only input being read data. This is likely achieved using a state machine internally, as well as multiplexers and buffers to get information off the bus, meaning that the design uses mixed logic.

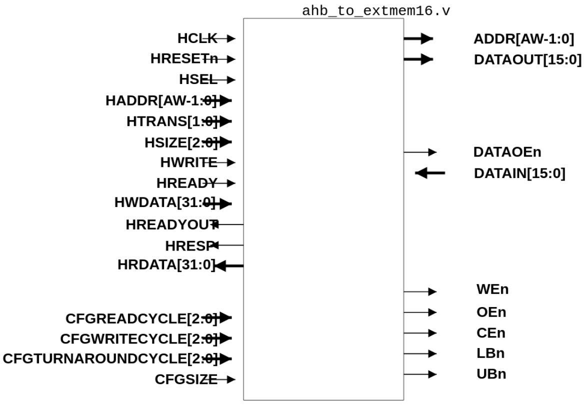

Figure 3: AHB to External Memory[6]

The address and write enable line can be passed on to the SRAM controller, however, Hamming codes are employed to protect the data from noise. This involves encoding incoming data and decoding outgoing data.

SRAM, static random access memory, is a type of volatile memory (data is only retained when powered) [4]. Comparing SRAM to DRAM, the key difference is the structure as SRAM consists of flip-flops requiring multiple transistors, which incurs a higher cost per GB when compared to DRAM. However, using flip-flips with feedback loops means SRAM need not be refreshed the way DRAM must, allowing SRAM to offer quick (single cycle) memory on-chip while also being lower power. As size per chip is limited, multiple SRAM ships are often used together.

As with most memory SRAM requires an address, data input, and write signal and produces only a data output. Most SRAM is also synchronous, requiring a clock.

Hamming codes are a type of error detection and correction code that allow for one erroneous bit to be detected and located, providing a small amount of noise protection. A Hamming code archives this by adding parity bits to blocks of data. Hamming codes only work with single-bit errors. [7] [8]

B)b VHDL Implementation

The design in part B)a can be improved in several ways. Improvement however is relevant, as it depends on what the design needs to be optimized for. Often systems will be optimized for speed, size, or power usage. If optimizing for speed parts of this system can be pipelined by adding registers that allow the SRAM controller to begin creating the next instruction while the data is being fetched. This will increase size and power usage somewhat, however.

I References

[1] L. Pittroff, *Tutorial: The role of jtag in system debug and test throughout the embedded system development lifecycle*, Oct. 2008. [Online]. Available: `https://www.embedded.com/tutorial-the-role-of-jtag-in-system-debug-test-throughout-the-embedded-system-development-lifecycle/` (visited on 05/15/2023).

[2] *Jtag test overview*, Nov. 2022. [Online]. Available: `https://www.corelis.com/education/tutorials/jtag-tutorial/jtag-test-overview/` (visited on 05/15/2023).

[3] R. Oshana, *Jtag: An introduction*, Jan. 2023. [Online]. Available: `https://www.embedded.com/introduction-to-jtag/` (visited on 05/15/2023).

[4] W. J. Dally, R. C. Harting, and T. M. Aamodt, *Digital Design using VHDL A Systems Approach.* Cambridge University Press, 2016.

[5] B. Walshe, *What is amba?* Dec. 2014. [Online]. Available: `https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/what-is-amba` (visited on 05/15/2023).

[6] "Cortex-m system design kit technical reference manual r0p0." (Feb. 22, 2023), [Online]. Available: `https://developer.arm.com/documentation/ddi0479/b/Basic-AHB-Components/AHB-to-external-SRAM-interface` (visited on 05/15/2023).

[7] J. R. Barry, *Digital communication.* eng, 3rd ed. / John R. Barry, Edward A. Lee, David G. Messerschmitt. Boston, Mass. ; Kluwer Academic, 2004, ISBN: 0792375483.

[8] U. Sani and I. Shanono, "Design of (7, 4) hamming encoder and decoder using vhdl.," Sep. 2015.